Intrusion Detection with OMNeT++

Bazara I. A. Barry
University of Khartoum
Faculty of Mathematical Sciences
Information Technology Research & Development Center
Network and Information Security Group

bazara.barry@gmail.com

ABSTRACT

Network simulators serve a variety of purposes. Compared to the cost, time, and effort involved in setting up an entire test bed containing different types of network devices, network simulators are relatively fast and inexpensive. Computer intrusions are occurring almost routinely and have become a major issue in our networked society. Every organization is faced by the big challenge of selecting an intrusion detection system and testing its abilities. Therefore, it is worthwhile to investigate the possibility of implementing and thoroughly testing intrusion detection systems using network simulators. In this paper, we report our experience with implementing and testing intrusion detection systems using OMNeT++ simulator. We highlight how OMNeT++ is harnessed to test and evaluate the intrusion detection system in terms of detection accuracy and performance.

Categories and Subject Descriptors

B.4.4 [**Performance Analysis and Design Aids**]: Simulation

General Terms

Design, Security

Keywords

intrusion detection, intrusion simulation, OMNeT++, performance evaluation.

1. INTRODUCTION

Primarily, an Intrusion Detection System (IDS) is concerned with the detection of hostile actions. IDSs are classified based on the detection approach to signature-based, anomaly-based, and specification-based systems.

Signature-based systems, using stored behavior patterns to identify and detect attacks, can detect known attacks accurately but are ineffective against previously unseen ones. Anomaly-based systems create a normal behavior model for a system using previously seen behaviors in the absence of attacks to classify any activities that violate the model as potential attacks. They can detect unknown attacks yet produce false alarms for legitimate but previously unseen behaviors. Specification-based intrusion detection, where manually specified program behavioral specifications are used as a basis to detect attacks, has been proposed as a promising alternative that combines the strengths of signature-based detection (accurate detection of known attacks) and anomaly-based detection (ability to detect unknown attacks).

Testing of Intrusion Detection Systems proves to be a challenging task due to the various considerations and players involved in the process. For instance, network administrators and security officers need to perform thorough tests on the products to compare their performance against other products. They also need to make sure that IDSs live up to the claims of vendors and expectations of customers. Studying and testing a new intrusion detection architecture against a variety of intrusive activities under realistic background traffic is an interesting and difficult problem. Such studies can be performed either in a real or simulated environment.

When simulating in real environments, many real users produce significant background traffic by using a variety of network services. This background traffic is collected and recorded, and intrusive activities can be emulated by running exploit scripts. In this approach, the traffic is sufficiently realistic, which eliminates the need to analyze and simulate normal user activities. However, the testing environment may be exposed to unexpected attacks which affect the accuracy of the results negatively. Due to the high risk of performing tests in a real environment, researchers have been tending to perform tests in simulated environments [18]. In this approach, realistic background traffic is generated using simulation tools, and attacks are injected accordingly.

A network simulator is a program that models the behavior of a network either by calculating the interaction between the different network entities using mathematical formulas, or actually capturing and playing back observations from a production network. The behavior of the network and the various applications and services it supports can then be observed in a test lab; various attributes of the environment can also be modified in a controlled manner to assess how the network would behave under different conditions.

In this paper, we report our experience with OMNeT++ simulator in testing a hybrid intrusion detection system that combines specification-based and signature-based approaches for attack detection and targets Voice over Internet Protocol (VoIP) environments. We start off by stating the hurdles that face testing of intrusion detection systems in real environments. Next, we introduce the main simulation concepts adopted by OMNeT++

and how they can be used to overcome the hurdles. A comparison against other simulators will be shown. We then discuss in some detail the components of the proposed hybrid IDS focusing on its features and advantages over other similar IDSs. Then we shed some light on how OMNeT++ is used to implement and test the IDS. Finally, we explore the features provided by the simulator to evaluate intrusion detection systems quantitatively before concluding the paper.

2. HURDLES OF INTRUSION DETECTION SYSTEM TESTING

The main aim of intrusion detection system testing is to quantitatively evaluate hit rate and probability of false alarms. Hit rate determines the rate of attacks detected correctly by an IDS in a given environment during a particular time frame, whereas probability of false alarms determines the rate of false alarms produced by an IDS in a given environment during a particular time frame. There are several challenges that face the testing of intrusion detection systems. Some of these challenges are:

1. **Collection of attack scripts:** An important aspect of the testing of any IDS is testing its ability to detect a wide range of attacks. Collecting a wide range of attack scripts and codes is a difficult task. Although many of these scripts and codes are available on the Internet, it entails a considerable time and effort to adapt them to the particular testing environment. Once the script of an attack is identified, it must be reviewed, automated, and smoothly integrated into the testing environment. Such tasks could be very challenging due to the fact that these scripts are developed by different people with different technical backgrounds to work in different environments.

2. **Use of different tools to launch and detect attacks:** Testing of intrusion detection systems usually involves two main phases. The first phase is to develop the intrusion detection algorithms and architecture using a specific tool. The second phase is to develop the attacks and scenarios necessary to test the system using a different tool. This separation of tools creates complications when it comes to integrating these tools to work together into the specific testing environment.

3. **Generation of background traffic:** Most IDS testing approaches can be classified in one of four categories with regard to their generation of background traffic [3]. Each of these categories has its advantages and disadvantages. In the following we summarize the four approaches and the challenges they pose:

   - **Testing using no background traffic:** In such a scheme, an IDS is set up on a host or network on which there is no activity. Then, computer attacks are launched on this host or network to determine whether or not the IDS can detect the attacks. This approach is useful for verifying that an IDS has signatures for a set of attacks and that the IDS can properly label each attack. Furthermore, testing schemes using this approach are often much less costly to implement than the other approaches. However, such a scheme can neither say anything about false alarms, nor about the IDS ability to detect attacks at high levels of background activity [15].

   - **Testing using real background traffic:** This approach is very effective for determining the hit rate of an IDS given a particular level of background activity. Hit rate tests using this technique may be well received because the background activity is real and it contains all of the anomalies and subtleties of background activity. However, this approach could be ineffective in determining false alarm rates. It is virtually impossible to guarantee the identification of all of the attacks that naturally occurred in the background activity, which hinders false alarm rate testing. It is also difficult to publicly distribute the test since there are privacy concerns related to the use of real background activity [5].

   - **Testing using sanitized background traffic:** In this approach, real background activity is prerecorded and then sanitized to remove any sensitive data. This sanitization is performed to overcome the political and privacy problems of using, analyzing, and distributing real background activity. Then, attack data are injected within the sanitized data stream. Attack injection can be accomplished either by replaying the sanitized data and running attacks concurrently or by separately creating attack data and then inserting these data into the sanitized data. The advantage of this approach is that the test data can be freely distributed and the test is repeatable. However, sanitization attempts may end up either removing much of the content of the background activity thus creating a very unrealistic environment, or removing information needed to detect attacks [6].

   - **Testing by generating background traffic:** In this scheme, a test bed or simulated network is created with hosts and network infrastructure that can be successfully attacked. The simulated network includes victims of interest with background traffic generated by complex traffic generators that model the actual network traffic statistics. An advantage of this approach is that the data can be distributed freely since they do not contain any private or sensitive information. Another advantage is that we can guarantee that the background activity does not contain any unknown attacks since we created the background activity using the simulator. Therefore, false alarm rates using this technique are well received. Lastly, IDS tests using simulated traffic are usually repeatable since one can either replay previously generated background activity or have the simulator regenerate the same background activity that was used in a previous test [1].

3. OMNeT++ SIMULATOR

OMNeT++ is an object-oriented discrete event simulation tool that uses a modular structure. It may be used for traffic modeling of telecommunication networks, protocol modeling, and evaluating performance aspects of complex software systems among other things.

3.1 OMNeT++ Simulation and Modeling Concepts

An OMNeT++ model consists of hierarchically nested modules, which communicate through message passing. OMNeT++ models are often referred to as *networks*. The top level module is the *system module*. The *system module* contains sub-modules which can also contain sub-modules themselves. The structure of models is shown in figure 1. The depth of module nesting is not limited, which allows the user to reflect the logical structure of the actual system on the model structure. Modules that contain sub-modules are termed *compound modules*, whereas *simple modules* lie at the lowest level of the module hierarchy. Simple modules contain the algorithms in the model.

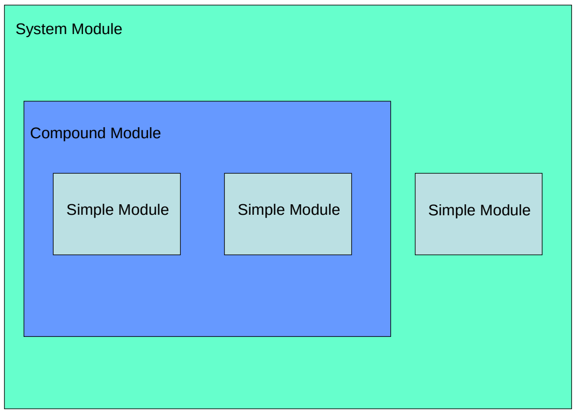

Figure 1. OMNeT++ Model Structure.

As previously mentioned, Modules communicate by exchanging messages. In an actual simulation, messages can represent frames or packets in a computer network and can contain arbitrarily complex data structures. Messages can arrive from another module or from the same module. When a message arrives from the same module, it is called a *self-message* and is usually used to implement timers. Simple modules can send messages either directly to their destination or along a predefined path, through *gates* and *connections*.

Gates are classified into output and input gates. Output gates are the interfaces through which messages are sent out, whereas input gates are the interfaces through which messages arrive. Connections are the links used to connect gates. Connections can be assigned three parameters, which facilitate the modeling of communication networks. These three parameters are *propagation delay* (which is the amount of time the arrival of the message is delayed by when it travels through the channel), *bit error rate* (which specifies the probability that a bit is incorrectly transmitted and allows for simple noisy channel modeling), and *data rate* (which is specified in bits/second and used for calculating transmission time of a packet).

OMNeT++ uses two programming languages, namely NED (Network Description) Language and C++. NED language is used to describe the model structure and the topology of a network and its modules. A network description may consist of a number of component descriptions that can be reused in another network description, which facilitates the modular description of a network. NED files can be created with any text-processing tool and have a human-readable textual topology. On the other hand, C++ is used for the actual implementation of the simple modules such as messages and queues. The full flexibility and power of the programming language can be used, supported by the OMNeT++ simulation class library. The simulation programmer can freely use C++ object-oriented concepts (inheritance, polymorphism, etc) and design patterns to extend the functionality of the simulator.

A simulation program in OMNeT++ is built from: (1) NED files (*.ned*) which describe the module structure with parameters and gates, (2) message definitions (*.msg* files) which define message and packet types and structures, (3) simple module sources which are written in C++ (*.cc* and *.h*), and (4) initialization files (*.ini*) to set values to parameters defined in *.ned* files. Therefore, the design of a topology and the implementation of the modules that exist in the topology are separated [7].

A model network in OMNeT++ consists of nodes that are connected by links. The nodes represent network components (such as hosts, routers, and switches), whereas the links represent channels and network connections (such as Ethernet). OMNeT++ follows the previously mentioned hierarchical approach to organize the building of networks at different levels. A network description in OMNeT++ consists of multiple levels to define channels with their characteristics, sub-networks or Local Area Networks (LANs) with their boundaries within the topology, and individual nodes with their attributes. Within a node, the traditional layered networking approach is followed to provide Link, Network, Transport and Application layer protocols and applications. The channels, simple modules, and compound modules of one network description can be reused in another network description, which puts OMNeT++ ahead of many network simulators that provide poor reusability only via copy-and-modify operations.

OMNeT++ support for TCP/IP protocols such as Internet Protocol (IP), Internet Control Message Protocol (ICMP), User Datagram Protocol (UDP), and Transmission Control Protocol (TCP) started with the Internet Protocol Suite (IPSuite), and has culminated in the more recent INET Framework. Both IPSuite and INET framework have been faithful to the TCP/IP protocol suite with its layered approach. Several research groups at the University of Karlsruhe developed MMSim [4] which is a model to simulate Voice over IP (VoIP) protocols using OMNeT++. The MMSim model provides support for Session Initiation Protocol (SIP), Real-time Transmission Protocol (RTP), and Real-Time Streaming Protocol (RTSP). The modularity that distinguishes OMNeT++ is reflected on the modeling of networking protocols, where all components of protocols are divided into a number of different modules, and each module can have several parameters. The actual details of each protocol are implemented in C++ programming language, where every major operation of the protocol is implemented as a *member function* in the class files that represent the protocol. All implementations of protocols follow the specifications detailed in relevant Request for Comment (RFC) documents.

3.2 Comparison with Related Simulators

In order for the reader to appreciate why OMNeT++ has been chosen for our implementation and to show the simulator advantages and strengths, we compare it against two popular simulators, namely, Ns-2 [14] and OPNET [9]. Ns-2 is popular in academia for its extensibility (due to its open source model) and plentiful online documentation. OPNET Modeler is the industry's leading simulator specialized for network research and development.

Despite the remarkable similarities between Ns-2 and OMNeT++, we attempt to show the advantages of the latter in terms of model management, support for hierarchical models, and debugging and tracing capabilities.

OMNeT++ models are easier to manage in the sense that they are independent from the simulation kernel. In OMNeT++, the simulation kernel is a class library that is easily distinguishable from the components written by programmers and researchers. After writing components (simple modules), programmers can link their executables with the simulation library with no need to modify OMNeT++ sources. In Ns-2 however, the need to modify source packages may arise due to the blurriness between simulation kernel and user models.

The hierarchical module structure in OMNeT++ helps to tackle model complexity. Any complex component can be implemented as one unit (simple module) or built out of several smaller components (compound module). On the other hand, creating complex components as a composition of several independent units in Ns-2 is a challenging task due to the flatness of models.

On the front of debugging and tracing, OMNeT++ provides *Tkenv* which is a graphical interactive execution environment that allows for the examination of simulation progress and changing of parameters in runtime. Conversely, Ns-2 lacks a capable Graphical User Interface (GUI) as *Tkenv*. Although Ns-2 provides the network animator (Nam) to view network simulation and packet traces, it falls short of providing convenient interaction with users.

Although OPNET is a leading commercial network simulator that includes a rich library for a wide spectrum of protocols, OMNeT++ can stand out in some aspects. For instance, OPNET assigns packet queues to input gates, and sent messages are buffered at the remote end of the link until they are received by the destination module. On the other hand, OMNeT++ gates do not have associated queues. Sent messages are placed in a data structure that is called the Future Event Set (FES). The FES is implemented as a binary heap to insure fast retrieval. OMNeT++'s approach is faster than OPNET's because it does not have the enqueue/dequeue overhead and also spares an event creation [8].

Due to their high importance and involvement in many applications, OMNeT++ provides solid support for Finite State Machines (FSMs). OMNeT++'s support for FSMs is very similar to OPNET's. OMNeT++ provides a dedicated class and a set of macros to build and manage FSMs efficiently.

3.3 How OMNeT++ Can Be Used to Overcome the Testing Hurdles

C++ programming language can be exploited efficiently to implement attacks, which alleviates the burden of integrating attack scripts and codes written in different programming languages and styles into the testing environment. The full flexibility and power of the programming language, supported by the OMNeT++ class library can be used to implement protocol-related attacks. The same powerful features can also be used to implement both attacks and detection algorithms without the need to switch tools or products. Furthermore, OMNeT++ can generate background traffic that is guaranteed to be free of unwanted attacks, which gives credibility to hit rate and false alarm tests, and the test scenario is usually repeatable.

4. VoIP INTRUSION DETECTION

We start this section by shedding some light on some basic VoIP principles as a prelude to the discussion on the architecture and components of our hybrid IDS. VoIP refers to the transmission of voice traffic over IP-based networks. Such a transmission and the associated services use several interacting protocols. The protocols covered by our IDS include VoIP application layer protocols: SIP for session initiation and RTP for data delivery.

Session Initiation Protocol (SIP) is a standard signaling protocol for VoIP and is described in Internet Engineering Task Force (IETF) RFC 3261. It addresses some important issues in setting up and tearing down sessions, such as user location, user availability, and session management. Its simplicity and versatility make it the choice of instant messaging, video conferencing, and multiplayer game applications among others. SIP uses other protocols such as Session Description Protocol (SDP) to describe the characteristics of end devices, Resource Reservation Protocol (RSVP) for voice quality, and Real-time Transport Protocol (RTP) for real-time transmission.

Elements in SIP can be generally classified into servers, endpoints, and routing nodes. SIP servers are the components responsible for various duties aiming at maintaining the service and enhancing it such as address resolution, registration, and call redirection. Endpoints (also known as User Agents UAs) are the devices capable of initiating or terminating a call. Routing nodes in VoIP environments have the capacity to connect VoIP networks to either other VoIP networks or circuit-switched networks.

The base SIP specifications define six types of request: the INVITE request, CANCEL request, ACK request and BYE request are used for session creation, modification, establishment, and termination; the REGISTER request is used to register a certain user's contact information; and the OPTIONS request is used as a poll for querying servers and their capabilities.

The session is initiated using the INVITE method. INVITE requests follow a three-way handshake model, which means that the user agent (UA), after receiving a final response to an INVITE request, must send an ACK request. After establishing a session, the users can send and receive data using RTP. The UA may send a CANCEL request to cancel an invitation to a session after it has sent the INVITE request. INVITE requests can also be sent within dialogs to renegotiate the session description. A session is terminated with a BYE request.

SIP is susceptible to many attacks such as Denial of Service (which includes scenarios like targeting a certain UA or server and flooding them with requests), tearing down sessions prematurely by sending fake BYE or CANCEL requests, and session hijacking by sending fake Re-INVITE requests [10]. With regard to RTP, attackers can inject artificial packets with higher sequence numbers, which causes the injected packets to be played in place of the real ones. Flooding with RTP packets deteriorates the perceived Quality of Service (QoS) and may also cause phones dysfunctional and reboot operations [13].

4.1 System Architecture and Components

The proposed architecture of the hybrid host-based intrusion detection system is shown in figure 2.

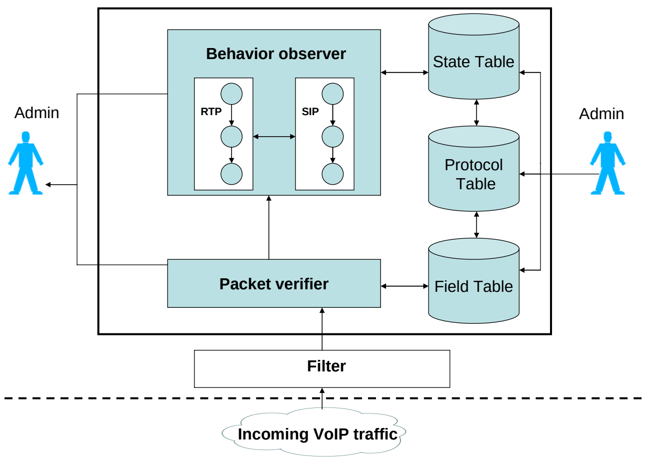

Figure 2. Hybrid Intrusion Detection System Architecture.

The *filter* receives the incoming traffic and classifies it into signaling (SIP) and media (RTP) packets. The *packet verifier* receives packets from the *filter* and examines them in terms of size and structure. Too big or malformed packets are rejected by the *packet verifier* in order not to deplete the processing power of the endpoint. Individual header fields of the packet are examined to check if they comply with the protocol specifications and whether mandatory fields are present. Then the system retrieves all the records of the field from the *field table* to perform signature detection for potential suspicious patterns associated with the field. Multiple records in this table can be used to form a signature that spans across many fields and protocols. The main fields of this table and their descriptions are: **Protocol ID**: uniquely identifies each protocol. **Field ID**: uniquely identifies the field of protocol header. **Field Name**: contains a name given to the field. **Description**: shows the function of the field. **Type**: contains the field data type. **Pattern**: field usually contains suspicious patterns the administrator is interested in detecting. **Stand-Alone Pattern**: A Boolean field to identify whether the above-described pattern forms an attack on its own or as part of other fields. **Impact**: The effect of the attack on the system. If approved, packets are sent to the *behavior observer*.

The *behavior observer* keeps track of the session and whether it progresses according to specifications. This session awareness is achieved by keeping Extended Finite State Machines (EFSMs) for the protocols involved to guard against any unacceptable behavior that violates proper protocol semantics. This way, unknown attacks can be detected by the *behavior observer*. Each protocol EFSM is provided with state variables to hold the values of header fields in incoming packets. A protocol EFSM is also provided with getter functions, so that other protocol EFSMs can get values of header fields and protocol state, which benefits the system in terms of detection accuracy.

When reaching a certain state in the EFSM, the system retrieves all the records of that state from the *state table* to perform further checks on semantics violations. Each record in this table represents a state in the protocol's EFSM. The main fields in the table are: **Protocol ID**: uniquely identifies each protocol. **State ID**: uniquely identifies a state in the protocol EFSM. **State Name**: contains a name given to the state. **Description**: describes the state and the system upon reaching it. **Threshold**: Identifies the upper limit for the number of requests that can be received at this state. **Time Unit**: Denotes the period of time during which the threshold is measured. **Timer**: value for a timer that can be used at the state. **Recommended Action**: a procedure that should be executed by the system upon reaching the state to detect potential attacks. **Impact**: The effect of the attack on the system. The *state table* follows State Transition Analysis (STA) techniques which provide a method of representing the sequence of actions that the attacker performs to achieve a security violation. A major advantage of using this technique is its ability to foresee an incoming penetration based on the current system state. The *state table* provides special procedures that are associated with its records to deal with expected attacks and penetrations. The *protocol table* is an auxiliary table that contains high-level information on the protocols and is used for organizational purposes. Clearly, detecting and reporting attacks take place in real-time.

As can be gathered from the abovementioned description, our design adopts some advanced intrusion detection techniques. Firstly, our architecture provides a stateful and cross-protocol detection in specification-based and signature-based modules. The *behavior observer* performs stateful detection by keeping the EFSMs of all the involved protocols and assembling state from multiple packets. It also performs cross-protocol detection by providing external interfaces between protocol EFSMs in the form of callable functions which return values of important protocol state variables. The *field table* has the ability to store signatures that cross protocol boundaries. Furthermore, the *state table* follows the progress of protocol sessions carefully providing stateful detection. The special procedures stored in the table have the ability to perform cross-protocol detection. Secondly, the design of the database tables is simple and clean. This advantage is achieved by separating the anomalies in protocol traffic from specific attacks. Thirdly, our design maintains a reasonable balance between database normalization and performance. We provide a less normalized database (two levels of hierarchy) with more attributes per table. A signature in our database is entirely stored in a single table (either *field* or *state table*), which reduces retrieval time significantly. Fourthly, our signature database can thwart obfuscation attempts made by attackers to evade detection by representing attacks in the *state table* using a higher-level and audit record independent representation.

4.2 Comparison with Related IDSs

Several IDSs have been proposed to meet the special needs of VoIP environments. SCIDIVE [19] is a stateful, and cross-protocol IDS for VoIP. SCIDIVE can be considered a signature based detection system rather than an anomaly based system. As mentioned previously, signature-based systems lack the ability to detect new and novel attacks, and the rule database needs to be updated on a regular basis following new attacks. This limitation is addressed by vIDS [13]. Instead of relying entirely on a rule database, vIDS is based on interacting protocol state machines. However, all the attacks used to test the efficiency of vIDS were known attacks and had to be encoded in the system as attack patterns. The capabilities of vIDS in detecting attacks based on normal behavior specifications were not shown. Moreover, the design of vIDS covers the issues relating to protocol-semantics anomaly detection, while not addressing protocol-syntax anomaly detection.

vFDS [12] is an online statistical detection mechanism designed for VoIP systems. vFDS relies on pure statistical anomaly approaches which affect its sensitivity negatively. In addition, vFDS is limited to detecting flooding attacks. Our design provides a combination of specification-based and signature-based detection techniques to bring the false alarm rate to its lowest level. It also addresses syntax and semantics-related issues to cover a wider range of attacks.

5. IMPLEMENTATION AND TESTING USING OMNeT++

5.1 Implementation of Attacks and Detection Components

Attacks that target networked environments take advantage of vulnerabilities in networking protocols. Such attacks can be classified into (1) message flow attacks which are used by attackers to exploit vulnerabilities in the flow of messages used by protocols, (2) parser attacks which aim at hampering proper parsing by constructing invalid messages, and (3) flooding attacks which are used by attackers to deny legitimate users access to network resources.

In light of the above, we classify the implemented attacks based on the targeted protocols for implementation reasons. As mentioned earlier, protocols in OMNeT++ are implemented in an object-oriented manner as classes using C++ programming language. The main operations of each protocol are implemented as member functions in the class files. Therefore, we follow the same concept and implement protocol-related attacks as member functions in the class files that represent the protocol.

Detection algorithms are implemented in some of the member functions that perform tasks related to the protocol operation. For instance, *handleMessage()* method, which is a member function responsible for handling messages coming to the protocol module, is used for the implementation of the detection algorithms responsible for checking the validity of the incoming packets and compliance of sessions with specifications. The protocol EFSMs, which form the main part of our *behavior observer*, are implemented in this method.

All attacks are given identification numbers, which are stored in a system text file. The code that launches attacks (calls the member function that represents the attack) chooses a number randomly from the range of the identification numbers, and launches the associated attack accordingly. Furthermore, the attack launching code itself is activated in the endpoints based on a randomly selected number that should exceed a certain threshold. This technique guarantees that the majority of the simulated network background traffic remains benign. Such techniques are made possible by the random number generation features provided by OMNeT++. OMNeT++ enjoys the support of several Random Number Generators that can be configured in the initialization files.

Events in OMNeT++ simulator environment can be controlled to occur at a specific time. Message/event related functions can be used to send messages to other modules, schedule an event, or delete a scheduled event. In our implementation, we use *send()*, *scheduleAt()*, and *cancelEvent()* methods to send packets, schedule, and cancel events respectively. The abovementioned methods provide different flavors with different parameter settings. This feature facilitates the detection and launching of attacks that require accurate timing such as flooding attacks, and message flow attacks.

Message manipulation functions provided by protocol modules allow for creating malformed packets and launching parser attacks easily. The simulator library contains various functions to set the value of different fields, and the length of the entire message. For example, *setLength()* method allows for the creation of packets with lengths that go beyond the protocol specifications. Similar functions can be used to get the value of message fields to perform detection. Such functions are used to build our *packet verifier*.

MMSim module provides interaction between SIP and RTP which makes cross-protocol detection at the application layer possible. RTP attributes can be captured by SIP through a specialized function that can be called from SIP module.

On the other hand, C++ streams, which are associated with files, are used to emulate our signature database. All protocols, protocol header fields, and protocol states are given identification numbers, and all this information alongside relevant detection information is stored in the abovementioned files which act as database tables.

System files can also be used to aid the IDS in terms of performing stateful detection. Values of header fields of incoming packets are stored in temporary system files associated with sessions. Such files are named in a way that reflects the ID of the affiliated session, and the files contain records for the packets belonging to the active protocols of the session. This feature allows modules such as the *Field Table* to store signatures that span across multiple packets that could belong to different protocols. Since the relevant information is kept in these system files, the IDS can perform detection for the entire session or connection. Such system files get deleted automatically once the session is terminated.

5.2 Network Topology and Configuration

Figure 3 shows the simulated network topology. Our simulated network comprises two domains each with a Proxy and Registrar Server. Proxy servers are elements that route SIP signaling requests to servers and SIP signaling responses to clients. A registrar is a server that accepts client’s requests to register in a certain domain and places the information it receives in those requests into the location service for that domain. Each domain also contains a set of User Agents (endpoints) which are connected to the servers by a 10Base-T Ethernet.

We use the Audio/Video profile with minimal control (RTP/AVP), with UDP as the underlying protocol. An application profile describes how audio and video data may be carried within RTP. Our payload type is static with the identification number 10 and has the encoding L16. The payload type defines how a particular payload is carried in RTP. The clock rate, which is used to generate RTP timestamps, is 44100 Hz and the number of transmission channels is 2. Endpoints in a domain make calls to other endpoints in the other domain randomly and without predefined durations. The abovementioned parameter setting is recommended as one of the standard operating parameter settings for audio encoding and payload type [11].

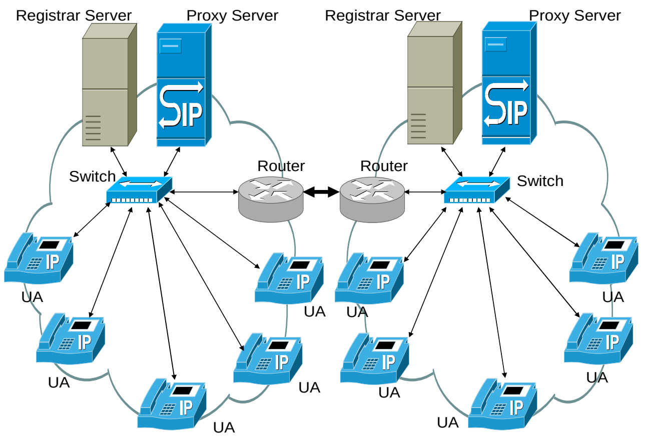

Figure 3. Simulated Network Topology.

Our IDS is installed on all endpoints and servers in both domains. The Internet connection between the two domains is assumed to have a delay of 40 ms and a packet loss of 0.2%. Such values for delay and packet loss are acceptable by most network Service Level Agreements (SLAs) for backbone providers [16].

It is important to mention that all of the abovementioned features which allow OMNeT++ to create real-like networking environments can easily be configured in the NED and initialization files of OMNeT++ without the need to compile the sources.

5.3 Traffic Generation

We aim at proving that the intrusion detection system can operate under stressful network conditions, adds little overhead to the network, and is robust. To do this, performance tests are conducted on the simulated network using a high-load scenario. We run the experiment under the high load for five different times. Each run lasts for 120 minutes which gives as an overall simulation time of 20 hours. The results, which will be shown in the next section, are averaged across the different runs and taken with and without the operation of the IDS to observe the difference. We use background traffic sent at a frequency of 1.5 calls per 1 second. Figure 4 shows the calls captured at the proxy server in one of the domains.

OMNeT++ provides numeric parameters that can be set in the initialization files to return random numbers distributed uniformly or from various distributions. For example setting a parameter to *truncnormal(3, 0.7)* would return a new random number from the truncated normal distribution with mean 3.0 and standard deviation 0.7 every time the parameter is read from the C++ code. Such a feature is useful for traffic generation and specifying inter-arrival times for generated calls. We use the Uniform distribution for our traffic generation.

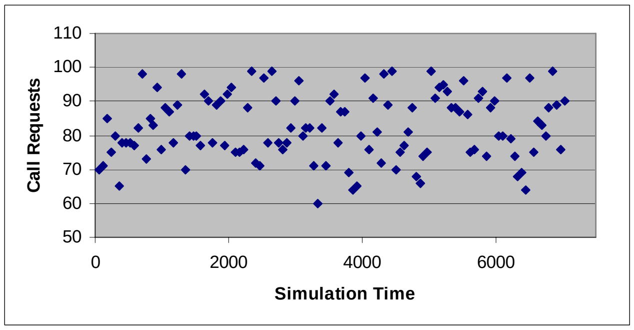

Figure 4. Call Requests at a Proxy Server under High-Load.

6. IDS EVALUATION AND EXPERIMENTAL RESULTS

In this section we discuss how the simulator is used to evaluate the performance of the IDS. Our discussion will revolve around two axes, namely, the IDS coverage and runtime impact.

6.1 IDS Coverage

Assessing the coverage of intrusion detection systems is a challenging task with many ramifications. The coverage of any intrusion detection system depends on the attacks that the IDS can detect under ideal conditions. The number of dimensions that form each attack makes the assessment difficult. Each attack has a particular goal and works against particular software. Attacks may also target a certain version of a protocol used or a particular mode of operation. Different sites may consider some attacks more important than others, which affects the assessment greatly. For instance, E-commerce sites may be very interested in detecting distributed denial of service attacks, whereas military sites may pay a great deal of attention to surveillance attacks.

We list in table 1 all the attacks implemented to test the system along with the protocols they target and the effect they have on the attacked system. We implement six attacks using the simulator to demonstrate the functionality of the intrusion detection system at the application layer. Some of these attacks can be found in classifications such as the one released by VoIPSA for threats that VoIP systems are vulnerable to [17].

There are several dimensions that can be taken from table 1. It is important to realize the diversity of the attacks implemented by the simulator to test the system in terms of the protocols involved and the effect they have. Some attacks are cross-protocol which forms another dimension. As shown in the table, the effect of the attacks varies widely. The attacks violate many of the security services that should be provided by systems such as availability, confidentiality, authentication, and data integrity. Therefore, we can safely say that OMNeT++ allows us to implement attacks that cover a wide range of protocols and security threat.

During the experiment, the IDS has managed to detect all the attack instances presented. Some of the attacks such as CANCEL and malformed packets were unknown to the IDS prior to the experiment. In other words, we did not encode any special signatures, and hence all detections for such attacks were based on normal behavior specifications. OMNeT++ can also be used to simulate false attacks. Such a feature allows us to plot Receiver Operating Characteristic (ROC) curves which show the detection rate versus false alarms per time unit. It is important to realize that the simulator's ability to implement and detect attacks is not confined to the attacks used during the experiment. The implemented attacks are meant to represent a wide range of security service violations and attack categories. The proposed intrusion detection components, which are implemented with OMNeT++, are capable of detecting other attacks that violate the syntax or semantics of protocols.

6.2 Runtime Impact

It is vital that any security measure to be implemented in a VoIP network does not impede the performance of the network. Quality of Service (QoS) is very important to the operation of VoIP networks. The implementation of various security measures in a VoIP network can introduce some complications that can degrade QoS. These complications range from delaying call setups to delaying delivery of data packets. In this section we show how OMNeT++ can be used to measure the impact of the IDS on the environment quantitatively. We will show some of the features provided by the simulator to measure various delays, packet loss, and memory consumption caused by the operation of the IDS.

OMNeT++ provides three important methods to return times associated with messages (packets), namely, *creationTime()* to return the message creation time, *sendingTime()* to return the message last sending time, and *arrivalTime()* to return the message last arrival time. Such values can be used to calculate various delays such as end-to-end delay (which refers to the time it takes for a voice transmission to go from its source to its destination), call setup delay (which refers to the period that starts when a caller dials the last digit of the called number and ends when the caller receives the last bit of the response), and processing delay (which is the time required by an endpoint or a server to process a message) among others. Figure 4 shows the end-to-end delay experienced by an endpoint in the network with and without our IDS installed. The figure shows end-to-end delay for individual RTP voice packets. Our IDS added about 2.6 milliseconds on average to the voice transmission delay. As shown in the figure, the overall delay remains considerably less than the upper bound of 150 milliseconds defined as the acceptable one-way delay for high voice quality [2]. The delay variation (jitter) remains at 2 milliseconds with a slight addition of 3 * 10^{-5} seconds by our IDS. Therefore, our IDS has a trifling impact on end-to-end delay.

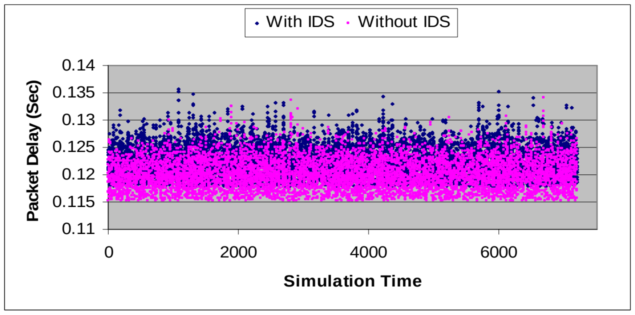

Figure 5. End-to-end delay.

Queues and their characteristics can be efficiently simulated in OMNeT++. The simulator's library provides a container class called *cQueue* that can hold objects of almost all types in OMNeT++ library. An important aspect associated with queues of network devices is packet loss. Packet loss at endpoints and servers could be the result of high sending rates especially in transmissions based on protocols that lack built-in transmission control mechanisms such as UDP, or processing spikes which mean that the CPU is spending too much time on some packets which has the consequence of missing subsequent ones. The packet loss rate at endpoints and servers can be affected by the operation of IDS. Figure 6 shows the packet loss rate at servers and endpoints queues with and without our IDS for various amounts of traffic. The packet loss rate with our IDS is only 0.02% higher than the rate without it on average. The overall packet loss remains at 0.04% on average, which is considerably less than the 1 percent level specified by many codecs as the upper limit.

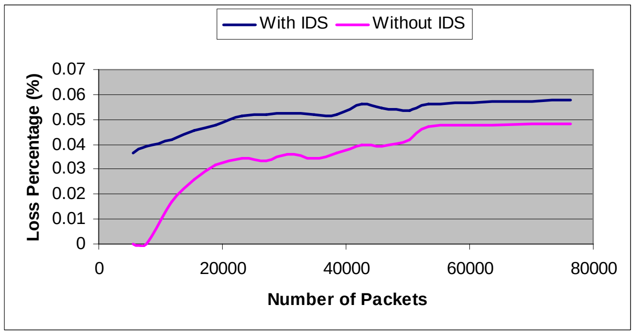

Figure 6. Packet loss.

Measuring the memory consumption of an intrusion detection system is vital in gauging its effect on the host. Some IDSs could exhaust all the available memory after a relatively short runtime, leaving the host with the possibility of crashing. We therefore use the functions provided by OMNeT++ class library and C++ programming language to identify the IDS's main data structures and add methods to track their size during simulation. Figure 7 shows the memory usage of the IDS at a server. The figure exhibits the gradual increase in memory consumption as call and session establishment requests arrive.

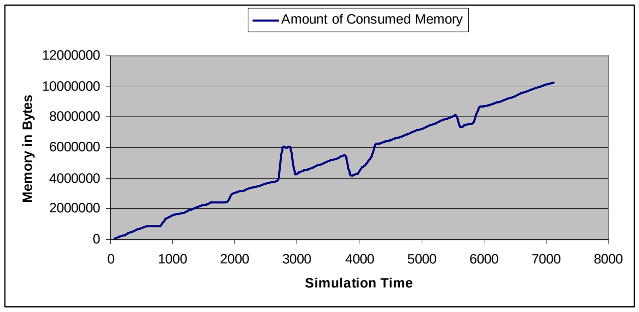

Figure 7. Memory consumption.

Memory consumption at the server starts at 96.2 KB and grows linearly till it reaches 3.8 MB as the simulation time passes the 40 minutes mark. The figure shows a surge in consumption that brings the amount of consumed memory to 6 MB. The surge can be attributed to a sudden increase in the number of connection and session establishments. Thereafter, the amount of consumed memory is decreased to remain around 4 MB as 1 hour of simulation time elapses. Afterwards, the figure shows a linear increase in memory consumption followed by a decrease before the overall consumption stabilizes around 10 MB. Such a figure is acceptable considering the plenty amounts of memory enjoyed by servers these days. When perceiving memory consumption results it is important to bear in mind that the experiment has been run using a high-load scenario whereby new calls are arriving every second and established calls deliberately linger to occupy memory for long durations of time.

7. CONCLUSION

In this paper, we have presented how OMNeT++ simulator can be used to implement and efficiently test an intrusion detection system suitable for VoIP environments. We have demonstrated how OMNeT++ is used as an evaluation framework that can be utilized to generate attacks and implement detection methodologies. The simulator has been successfully used to implement a novel intrusion detection architecture, and to collect various results aiming at assessing its performance aspects. We have clearly shown that the framework can be reliably used to test both the hit rate and false alarm rate of intrusion detection systems. Considering the availability and ease of use of OMNeT++ as an open source simulator and the hurdles that face traditional testing methods and tools, we believe our framework can form a solid base for future research in this area.

8. REFERENCES

[1] Durst, R., Champion, T., Witten, B., Miller, E., and Spagnuolo, L. 1999 Testing and Evaluating Computer Intrusion Detection Systems. *Communications of ACM, 42* (7), 53-61.

[2] International Telecommunication Union – Telecommunication Standardization Section Recommendation G.114: One-way Transmission Time. May 2003. Retrieved March 2008, from ITU web site: http://www.itu.int.

[3] Mell, P., Hu, V., Lipmann, R., Haines, J., and Zissman, M. 2003 An Overview of Issues in Testing Intrusion Detection Systems. Technical Report. NIST IR 7007, National Institute of Standard and Technology. Available: http://csrc.nist.gov.

[4] MMSim – Simulation of Multimedia Protocols using OMNeT++. Retrieved January 2008, from http://www.ibr.cs.tu-bs.de/projects/mmsim.

[5] Mueller, P. and Shipley, G. 2001 Dragon claws its way to the top. *Network Computing*, 45-67.

[6] National Laboratory for Applied Network Research 2003. NLAR Network Traffic Packet Header Traces. Available: http://pma.nlanr.net.

[7] OMNeT++ Simulator. Retrieved January 2008, from OMNeT++ web site: http://www.omnetpp.org.

[8] OMNeT++ User Manual. Retrieved October 2008, from OMNeT++ web site: http://www.omnetpp.org/doc/usman.html.

[9] OPNET Modeler. Retrieved June 2008, from OPNET web site: http://www.opnet.com.

[10] Poikselka, M., Mayer, G., Khartabil, H., and Niemi, A. 2004 *The IMS: IP Multimedia Concepts and Services in the Mobile Domain*. Wiley, Sussex, 278.

[11] Schulzrinne, H. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 1890, IETF Network Working Group. January 1996. Retrieved March 2008, from IETF web site: http://tools.ietf.org.

[12] Sengar, H., Wijesekera, D., Wang, H., and Jajodia, S. 2006 Fast Detection of Denial-of-Service Attacks on IP Telephony. In *Proceedings of IEEE Fourteenth International Workshop on Quality of Service*, (New Haven, CT, 2006).

[13] Sengar, H., Wijesekera, D., Wang, H., and Jajodia, S. 2006 VoIP Intrusion Detection Through Interacting Protocol State Machines. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, (Philadelphia, USA, 2006).

[14] The Network Simulator Ns-2. Retrieved March 2008, from Ns-2 web site: http://www.isi.edu/nsnam/ns/.

[15] The NSS Group 2003. Intrusion Detection System Group Test (Edition 4). Available: http://www.nss.co.uk.

[16] Voip-Info.org, *QoS*, 2004. Available: http://www.voip-info.org

[17] VOIPSA. VoIP Security and Privacy Threat Taxonomy, October 2005. Available: http://www.voipsa.org.

[18] Wan, T. and Yang, X. 2001 IntruDetector: A Software Platform for Testing Network Intrusion Detection Systems. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001)* (New Orleans, Louisiana, December 2001).

[19] Wu, Y., Bagchi, S., Garg, S., Singh, N. and Tsai, T. 2004 SCIDIVE: A Stateful and Cross Protocol Intrusion Detection Architecture for Voice-over-IP Environments. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)* (Florence, Italy, 2004).

9. APPENDIX A: TABLE 1. IMPLEMENTED ATTACKS WITH TARGETED PROTOCOLS AND EFFECT.

Attack Name	Brief Description	Protocols Involved	Effect
BYE Attack	A faked request sent by attackers to fool the parties involved in a session into tearing it prematurely.	SIP, RTP	Session Tear down
Re-INVITE Attack	A faked request sent by attackers to one of the parties involved in a session to fool it into redirecting the call to the attacker.	SIP,RTP	Session Hijacking
CANCEL Attack	A faked request sent by attackers to cancel a call attempt made by legitimate users.	SIP	Denial of Service
Malformed Messages	Malformed protocol messages created by attackers to hamper victim processing	All Protocols	Denial of Service
REGISTER Flooding	Overwhelming registrar servers with too many requests within a short time.	SIP	Denial of Service
Voice Injection	Injecting an alternative voice stream to one of the parties involved in a session.	RTP	Playing Artificial Stream